Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

Student names: Bonnesoeur Maxime, Gautier Maxime, Furrer Stanislas

Instructions: Update this file (or recreate a similar one, e.g. in Word) to prepare your answers to the questions. Feel free to add text, equations and figures as needed. Hand-written notes, e.g. for the development of equations, can also be included e.g. as pictures (from your cell phone or from a scanner). ***This lab is graded.*** *and needs to be submitted before the* ***Deadline : Wednesday 15-05-2019 Midnight. You only need to submit one final report for all of the following exercises combined henceforth.*** *Please submit both the source file (*.doc/*.tex) and a pdf of your document, as well as all the used and updated Python functions in a single zipped file called final_report_name1_name2_name3.zip where name# are the team member's last names. Please submit only one report per team!*

NOTE : The following exercises on Salamandra robotica are based on the research of [1], [2] and [3].

Swimming with Salamandra robotica – CPG Model

In this exercise you will control a salamander-like robot Salamandra robotica for which you will use Python and the dynamics simulator Webots. Now you have an opportunity to use what you've learned until now to make the robot swim (and eventually walk). In order to do this, you should implement a CPG based swimming controller, similarly to the architecture shown in Figure 1.

In the folder Webots you will find sub folders containing the simulated world file and Python codes describing the controller. ***NOTE :*** Do not change the relative positions of files within those folders.

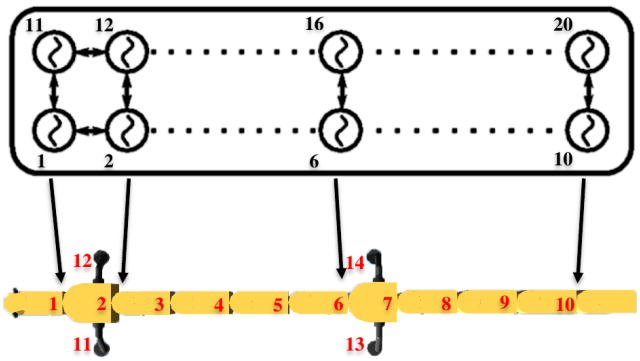


Figure 1: A double chain of oscillators controlling the robot's spine.

Code organization

- **Webots::worlds::cmc_salamander_#.wbt** - These are the world files which describe the worlds and allow to run the simulations. You can run a simulation by running this file with Webots. It also automatically loads the pythonController. Note that each of these files will also run the appropriate exercise_#.py such that you can run each exercise separately. Note that the simulation of the exercises may close immediately as they are not implemented yet. Only cmc_salamander_9_example.wbt will run for a few seconds before closing.
- **Webots::controllers::pythonController::exercise_#.py** - To be used to implement and answer the respective exercise questions. Note that exercise_example.py is provided as an

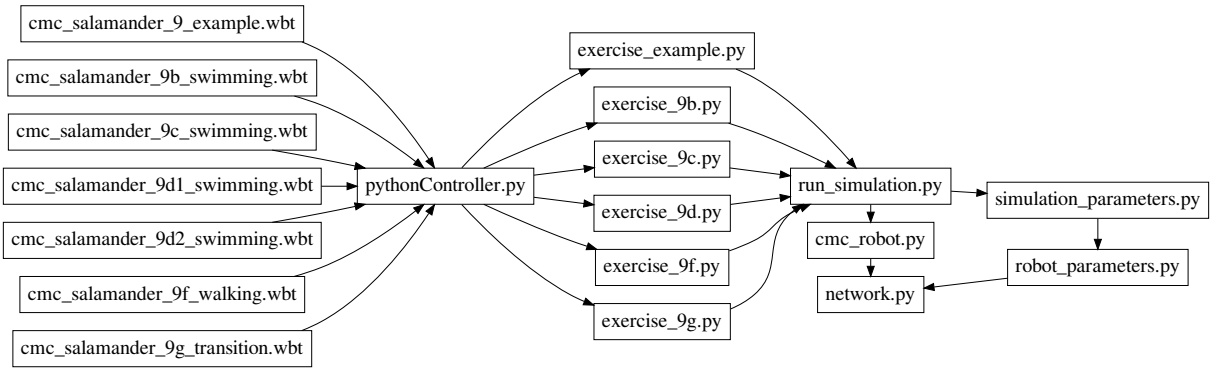


Figure 2: Exercise files dependencies. In this lab, you will be modifying `exercise1.py` and `pendulum_system.py`

example to show how to run a parameter sweep. Note that network parameters can be provided here.

- **Webots::controllers::pythonController::pythonController.py** - The main robot controller is implemented in pythonController.py. This file is the main file called by Webots during simulations and can call classes and functions from other files to control the robot and log data. This file is mainly used for calling the appropriate exercise_#.py file. Note that by default the simulation will close Webots when it finishes. If you do not want Webots to close, you can comment the following line: `world.simulationQuit(0)`.
- **Webots::controllers::pythonController::run_simulations.py** There is a run_simulation.py function which is provided for convenience to easily run multiple simulations with different parameters. You are free to implement other functions to run simulations as necessary.
- **Webots::controllers::pythonController::cmc_robot.py** - Contains the SalamanderCMC class, which is used for controlling and logging the robot.
- **Webots::controllers::pythonController::experiment_logger.py** - Contains the codes for logging the simulation. Feel free to modify this file to extend the logging capabilities. Note that the logging makes use of Numpy.savez to save the data.
- **Webots::controllers::pythonController::network.py** - This file contains the different classes and functions for the CPG network and the Ordinary Differential Equations (ODEs). You can implement the network parameters and the ODEs here. Note that some parameters can be obtained from pythonController.py to help you control the values.
- **Webots::controllers::pythonController::robot_parameters.py** - This file contains the different classes and functions for the parameters of the robot, including the CPG network parameters. You can implement the network parameters here. Note that some parameters can be obtained from SimulationParameters class in simulation_parameters.py and sent by exercise_#.py to help you control the values (refer to example).
- **Webots::controllers::pythonController::simulation_parameters.py** - This file contains the SimulationParameters class and is provided for convenience to send parameters to the setup of the network parameters in robot_parameters.py. All the values provided in SimulationParameters are actually logged in cmc_robot.py, so you can also reload these parameters when analyzing the results of a simulation.
- **Webots::controllers::pythonController::solvers.py** - This features fixed time-step solvers which will can are used by network.py for solving the ODE at each time-step. Feel free to switch

between the Euler and the Runge-Kutta methods. *You do not need to modify this files.*

- **Webots::controllers::pythonController::run_network.py** - By running the script from Python, Webots will be bypassed and you will run the network without a physics simulation. Make sure to use this file for question 9a to help you with setting up the CPG network equations and parameters and to analyze its behavior. This is useful for debugging purposes and rapid controller development since starting the Webots simulation with physics takes more time.
- **Webots::controllers::pythonController::plot_results.py** - Use this file to load and plot the results from the simulation. This code runs with the original pythonController provided.
- **Webots::controllers::pythonController::parse_args.py** - Used to parse command line arguments for run_network.py and plot_results.py and determine if plots should be shown or saved directly. *You do not need to modify this files.*
- **Webots::controllers::pythonController::save_figures.py** - Contains the functions to automatically detect and save figures. *You do not need to modify this files.*
- **Webots::controllers::pythonController::exercise_example.py** - Contains the example code structure to help you familiarize with the other exercises. *You do not need to modify this files.*

Prerequisites

Make sure you have successfully installed Webots by following the instructions outlined in Lab 7

Complete the tutorial and practice examples of Webots as outlined in Lab 7

Open the `Webots::worlds::cmc_salamander_9_example.wbt` file in Webots. This should launch the Salamandra robotica model in simulation world

Running the simulation

Now when you run the simulation, the Salamandra robotica model should float on the water with no errors in the Webots console dialog. At this point you can now start to work on implementing your exercises.

Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

Questions

The exercises are organized such that you will have to first implement the oscillator network model in run_network.py code and analyze it before connecting it to the body in the Webots world. Exercise 9a describes the questions needed to implement the oscillator models. After completing exercise 9a you should have an oscillator network including both the spinal CPG and limb CPG.

Using the network implemented in exercise 9a you can explore the swimming, walking and transition behaviors in the Salamandra robotica model using Webots and complete the exercises 9b to 9g.

9a. Implement a double chain of oscillators along with limb CPG's

Salamandra robotica has 10 joints along its spine and 1 joint for each limb. The controller

$$\dot{\theta}_i = 2\pi f + \sum_j r_j w_{ij} sin(\theta_j - \theta_i - \phi_{ij}) \tag{1}$$

$$\dot{r}_i = a(R_i - r_i) \tag{2}$$

$$\dot{q}_i = r_i(1 + cos(\theta_i)) - r_{i+10}(1 + cos(\theta_{i+10})) \text{ if body joint} \tag{3}$$

with θ_i the oscillator phase, f the frequency, w_ij the coupling weights, ϕ_{ij} the nominal phase lag (phase bias), r_i the oscillator amplitude, R_i the nominal amplitude, a the convergence factor and q_i the spinal joint angles.

1. Implement the double chain oscillator model using the functions `network.py::network_ode`. Test your implementation by running the network using `run_network.py`. For the network parameters check lecture slides (pay attention to different number of segments). You can also find more information in [3] (especially in the supplementary material). You can set all the network parameters in the `robot_parameters.py::RobotParameters`. To facilitate your work, you could start by only implementing the network for the body oscillators ($i = [0, ..., 19]$) and ignoring the leg oscillators ($i = [20, ..., 23]$). Refer to network::RobotState and robot_parameters.py::-RobotParameters for the dimensions of the state and the network parameters respectively.
2. Implement the output of your CPG network to generate the spinal joint angles according to equation 3. Implement this in the function `network.py::motor_output`. Verify your implementation in by running the Python file `run_network.py`. Use the functions in `plot_results.py` to report your spinal joint angles q_i.
3. Implement a drive and show that your network can generate swimming and walking patterns similarly to [3].

   Hint: The state for the network ODE is of size 48 where the first 24 elements correspond to the oscillator phases θ_i of the oscillators and the last 24 elements correspond to the amplitude r_i. The initial state is set in the init of network.py::SalamanderNetwork.

Answer By using the function 'run_network.py', we were able to test our parameters and obtain the following behavior for the walking and swimming pattern:

Swimming (drive = 4) By choosing a drive of 4, the robot adopt a swimming behavior. In this configuration, the legs are not moving as we can see in Figure 4 where the amplitude of the legs in red is equal to zero. Also, the amplitude of the oscillations of the spinal joints is quite high with a high frequency of oscillations to be able to go through an aqueous medium as can be seen in Figure 3

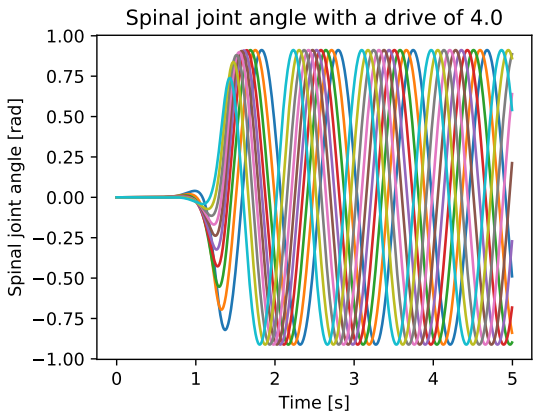


Figure 3: Spinal joint angle with a drive of 4

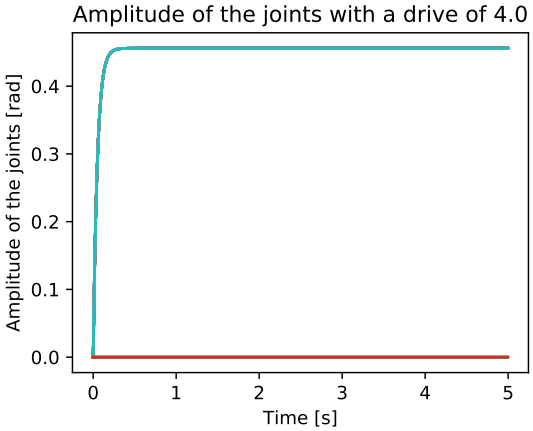


Figure 4: Amplitude of the joints for a drive of 4

Walking (drive = 2) By choosing a drive of 2, the robot adopts a walking behavior. In this configuration, the legs are moving as we can see on Figure 6 where the amplitude of the legs in red is not zero anymore. Also, the amplitude of the joints is lower this time and oscillate with a slower frequency as can be seen on figure 5. This is usual for the walking pattern were the robot becomes slower and does not rely anymore on a high oscillation to move itself.

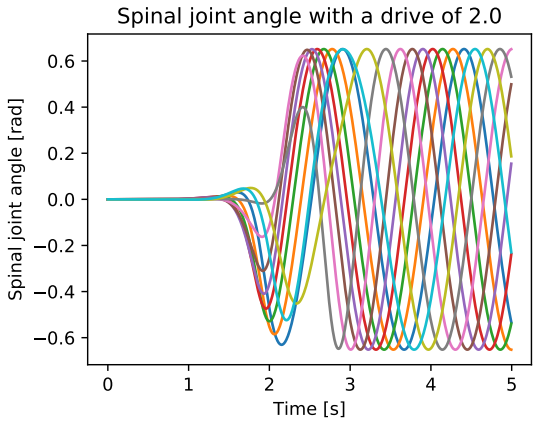


Figure 5: Spinal joint angle with a drive of 2

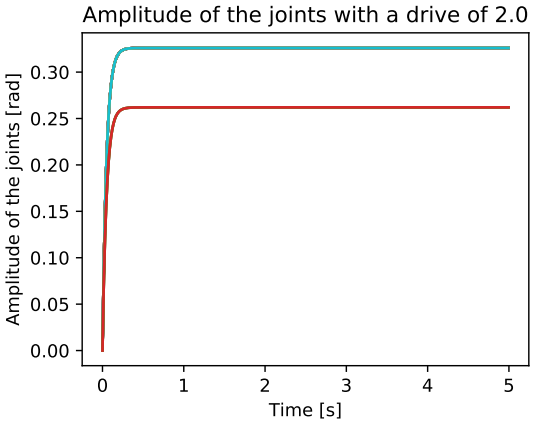


Figure 6: Amplitude of the joints for a drive of 2

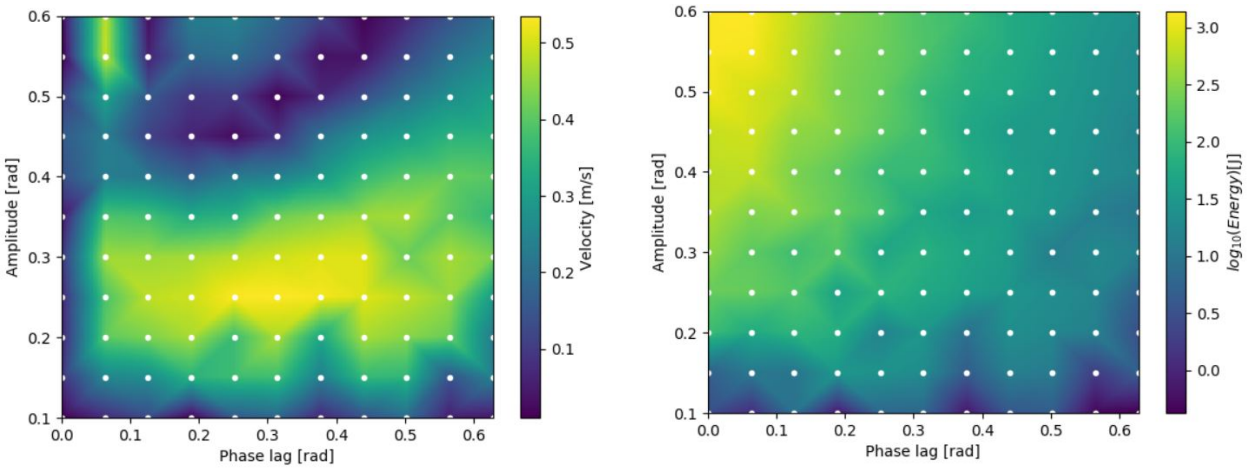
9b. Effects of amplitude and phase lags on swimming performance

Now that you have implemented the controller, it is time to run experiments to study its behaviour. How does phase lag and oscillation amplitude influence the speed and energy? Use the provided run_simulation.py::run_simulation() to run a grid search to explore the robot behavior for different combinations of amplitudes and phase lags. Use plot_results.py to load and plot the logged data from the simulation. Feel free to extend the logging in cmc_robot.py to show additional measurements if necessary. Include 2D/3D plots showing your grid search results and discuss them. How do your findings compare to the wavelengths observed in the salamander?

Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

- **Hint 1:** To use the grid search, check out the function run_simulation.py::run_simulation() and the example provided in exercise_example.py. This function takes the desired parameters as a list of SimulationParameters objects (found in simulation_parameters.py) and runs the simulation. Note that the results are logged as simulation_#.npz in a specified log folder. After the grid search finishes, the simulation will close, you can remove this feature by commenting world.simulationQuit(0) in pythonController.py::main().
- **Hint 2:** An example how to load and visualise grid search results is already implemented in plot_results.py::main(). Pay attention to the name of the folder and the log files you are loading. Before starting a new grid search, change the name of the logs destination folder where the results will be stored. In case a grid search failed, it may be safer to delete the previous logs to avoid influencing new results by mistake.
- **Hint 3:** Estimate how long it will take to finish the grid search. Our suggestion is to choose wisely lower and upper limits of parameter vectors and choose a reasonable number of samples. To speed-up a simulation, make sure to run Webots in a fast mode.
- **Hint 4:** Energy can be estimated by integrating the product of instantaneous joint velocities and torques. Feel free to propose your own energy metrics, just make sure to include the justification.

Answer



(a) Velocity *(b) Energy*

Figure 7: Velocity and Energy with varying amplitude and phase lag

The oscillator synchronizes at the fixed point

$$\rho_\infty = arcsin(A) + \phi_{12}$$

$$A = \frac{2\pi(v_2 - v_1)}{R_2 w_{12}}$$

This equation has no solution if A>1, one solution if A=1 and two solution if A<1. If A<1 the stability is defined by the sign of the function below. The fixed point is stable if this quantity is negative, and unstable if it is positive.

$$\text{If A=1 } \rho_\infty = \pm\frac{\pi}{2} + \phi_{12}$$

$$\text{If A<1 Sign Function : } R_2 w_{12} cos(\rho_\infty - \phi_{12})$$

From those equation we see that if we decrease the amplitude too much the factor A will increase above and hence the oscillator will drift. In the case of the phase lag, this value will only influence the stability through the equation above.

We run this grid search with a drive of 4.0, amplitudes varying from 0.1 to 0.6 and phase lag varying from 0 to $\frac{\pi}{5}$. Higher phase lags caused the salamander to either fold on itself, oscillate along the spine without moving, move backwards or oscillate between forward and backward motion, all of which falsify our results. The velocity is calculated as the distance travelled over time, starting 4 seconds after the start of the simulation in order to compare the stable speed of each pair of parameter and avoid the initial acceleration that could falsify results.

In Figure 7b we can see a spike of energy for maximum amplitude and low phase lag, with the energy lowering as the amplitude decreases and the phase lag increases. This makes sense since higher amplitudes will require an higher amount of energy to move the salamander. We can also see that this region does not correspond to a high velocity, which means the salamander spends a lot of energy moving around and turning on itself without actually moving forward. This is due to this high amplitudes which make the robot turn too much at each oscillation. This means that a non negligible part of the energy of the robot is lost in an oscillating behavior which is not contributing to the movement.

An aberration of the simulation can be seen for an amplitude of 0.55 and a phase lag of $\frac{\pi}{5}$ where the salamander curls completely on itself on one side then on the next one. It expands tremendous amounts of energy in doing so but manages to move quite fast, albeit completely unnaturally. This type of behavior can be observed in simulation but would be both dangerous and unnecessary in real life since better speeds can be reached with much less effort.

As we can see in Figure 7a, the velocity is optimal for an amplitude of around 0.25 and a phase lag of around 0.32. Various combinations of phase lag and amplitudes around those values also work well as it approaches optimal velocity. We can also see that with a phase lag of 0 the salamander is almost unable to move due to have no oscillation that spread throughout its body without creating an united movement. It has the same behavior as mentioned previously, curling up in a circle on one side then the other, once again expanding a lot of energy for little forward motion.

Optimal behavior would be fast movement with minimal energy and this can be found in the same zone as mentioned previously. We can note that coordinates with higher velocity correspond to the coordinates with a higher energy since more energy is required to move faster. However with good calibrations of both parameters, high velocity can be reached with relatively low energy since the robot will reach a stable behavior where most of its energy is spent moving forward instead of to the sides. These values induce a very smooth movement that makes the salamander move fast at a low energy cost.

9c. Amplitude gradient

1. So far we considered constant undulation amplitudes along the body for swimming. Implement a linear distribution of amplitudes along the spine, parametrized with two parameters: amplitudes of the first (Rhead) and last (Rtail) oscillator in the spine (corresponding to the first and last motor). To do so, you can add a parameter amplitudes=[Rhead, Rtail] in simulation_parameters.py::SimulationParameters. Don't forget to modify robot_parameters.py::RobotParameters::set_nominal_amplitudes() and interpolate the amplitude gradient between values Rhead and Rtail within the function. Note that you can then provide this amplitudes parameter from exercise_9b.py.
2. Run a grid search over different values of parameters Rhead and Rtail (use the same range for both parameters). How does the amplitude gradient influence swimming performance (speed,

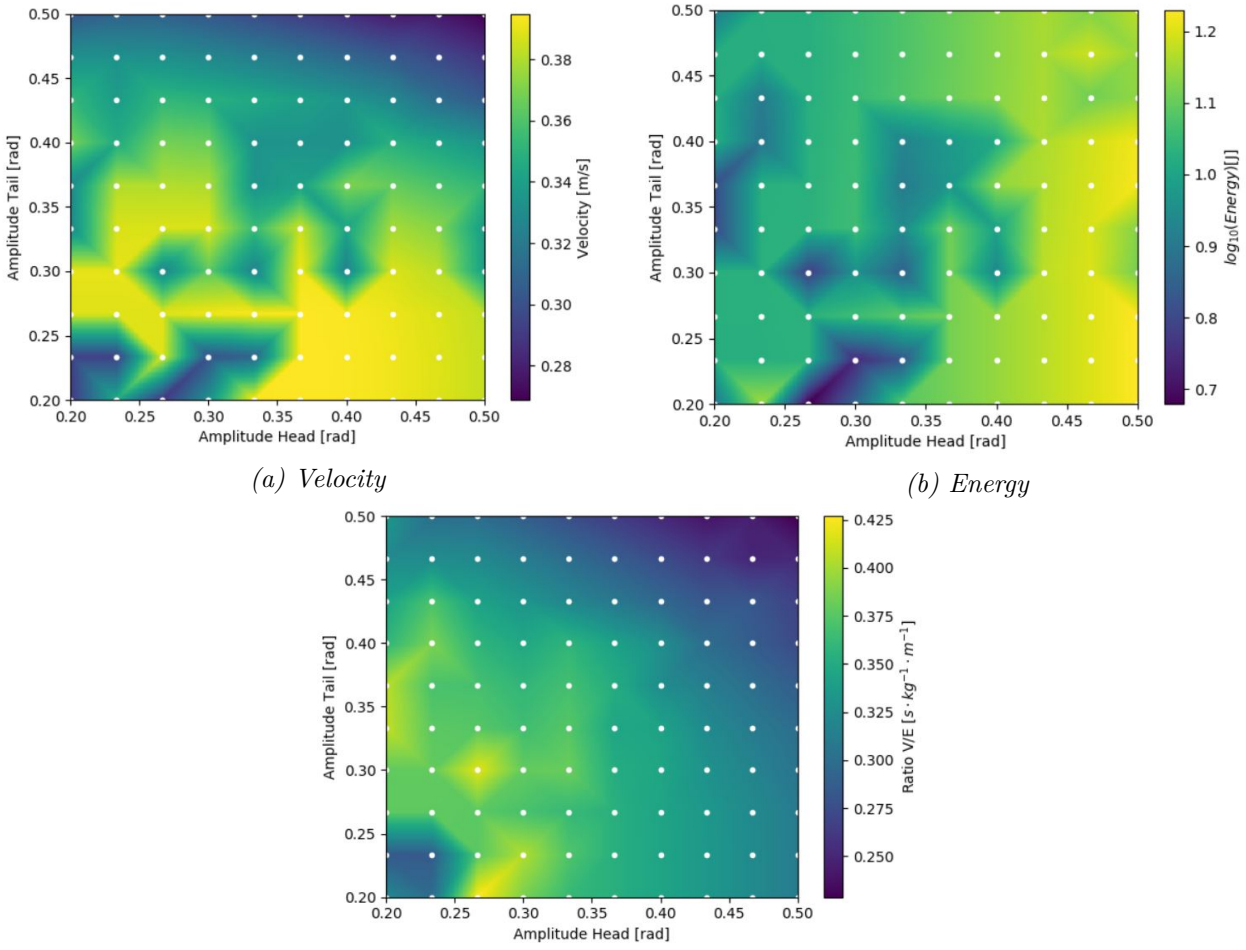
Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

energy)? Include 3D plots showing your grid search results. Do it once, for frequency 1Hz and total phase lag of 2π along the spine.

3. How is the salamander moving (with respect to different body amplitudes)? How do your findings in 2) compare to body deformations in the salamander? Based on your explorations, what could be possible explanations why the salamander moves the way it does?

Answer

2.



(a) Velocity

(b) Energy

(c) Ratio

Figure 8: Velocity and Energy with amplitude gradient

We ran the grid search for amplitudes varying from 0.2 to 0.5 since higher amplitudes caused saturation and lower amplitudes are not sufficient to make the robot move properly when the phase lag is 0, as can be seen in 7a.

We can see the influence of different amplitudes gradients on velocity and energy in Figure 8. First of all we can see a clear correlation between the velocity reached and the energy expended which is logical since lower velocities required less energy. We also see a general trend in both graphs. In Figure 8a we see that velocity increases as the tail amplitude decreases. This corresponds with results from 9b where amplitudes that are too high make the salamander curve too much and move more perpendicularly than forward. In Figure 8b we see that the energy increases with both amplitudes, but mainly the head amplitude. When the head amplitude is high, the movement starts at the head

and the rest of the body follows, which means the salamander has to pull the rest of its body which works well but requires a lot of energy. When the head amplitude is smaller, the movement starts lower in the body and the salamander uses its spine to push itself forward, which takes longer to accelerate but requires less energy.

Optimal behavior is a high velocity for a low energy, and when we look at the graphs we can see several combinations of amplitudes where the velocity and energy are lower. However the velocity is only slightly lower than the maximal where as the energy is much lower since it is on a logarithmic scale. We computed a ratio of velocity over energy to find the optimal behavior which can be seen in Figure 8c. We found was that optimal behavior prioritize low energy even if the velocity suffers slightly. This makes sense since the decreases of velocity is much smaller than the one of the energy which means that the relation between the two seems linear but with a slope lower than 1 : a lot of energy is necessary for a small increase in speed. Therefore, gaits with low energy should be prioritized since they are close in speed to more costly gaits but require very little energy.

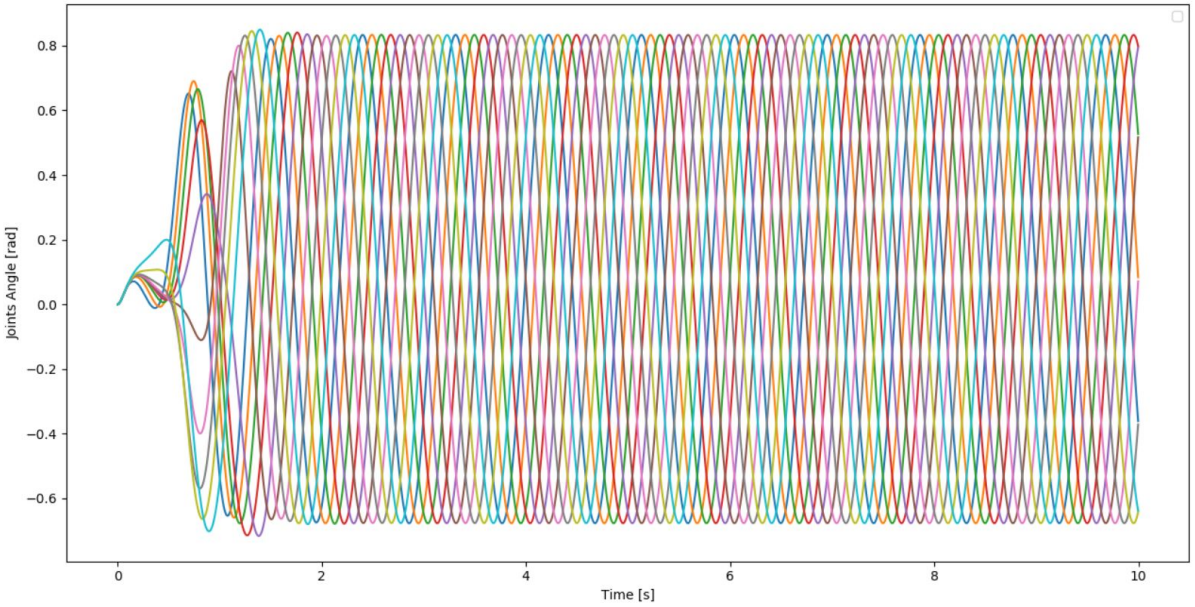
Although those gaits are not the fastest, they are the one let the salamander escape its static position the most easily by generating the smoothest movements.

3. The salamander moves forward but with obvious different amplitudes along the spine. Higher amplitudes makes the movements more sinuous so that the salamander travels more distance without actually getting much further from its original point. Basically, the optimal behavior is the one that is most fluid and harmonious and can be reached through several combinations.

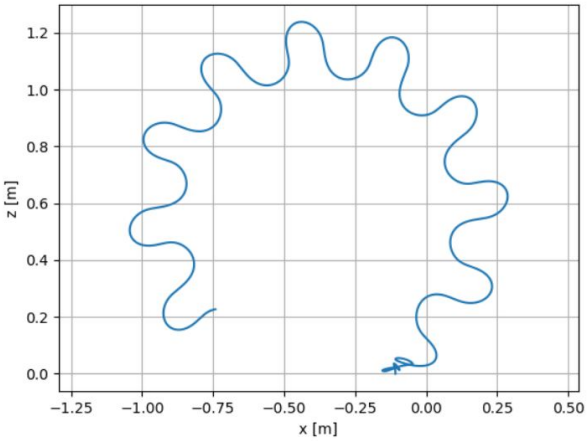
As mentioned above, with a higher amplitude in the tail, the salamander pushes forward using its body, whereas when the amplitude is higher in the head than in the tail, it must pull its body using its head, which requires a lot more energy.

9d. Turning and backwards swimming

1. How do you need to modulate the CPG network (network.py) in order to induce turning? Implement this in the Webots model and plot example GPS trajectories and spine angles.



(a) Joints Angles when turning



(b) Trajectory

Figure 9: Position and Trajectory when turning

To induce turning, we set a different drive for the two sides of the body for the setting of the nominal amplitude only. One side has a higher drive and hence produces a higher amplitude while the side with a lower drive has a smaller amplitude. The salamander will then turn to the side with the lowest amplitude. The difference in amplitude is determined by the value of the turn we wish to induce, the higher the value, the more the robot will turn. We used a turn value of 0.15 with a drive of 4.0 for Figure 9.
This configuration will induce a drive of $4.6 (= 4(1 + 0.15))$ on the left side and a drive of 0.6 $(= 40.15)$ on the right side of the salamander. This behavior, hence induce a turning in the right direction (the graph below is a view from the bottom of the robot hence theapparentt turning in the left direction).

2. How could you let the robot swim backwards? Explain and plot example GPS trajectories and spine angles.

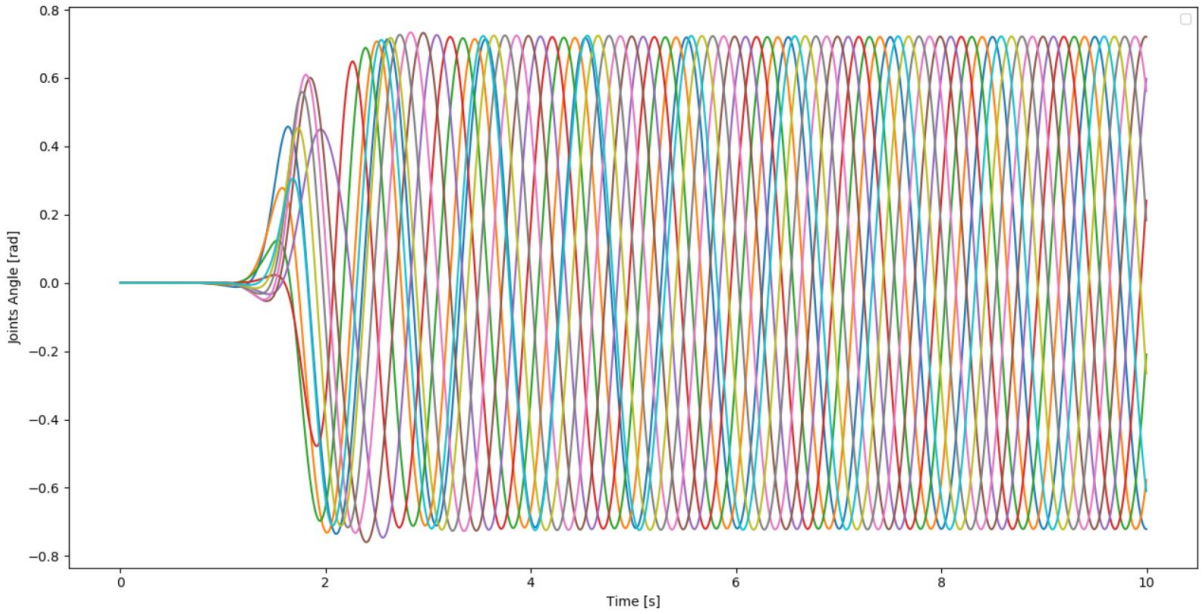


Figure 10: Joints Angles when going backward

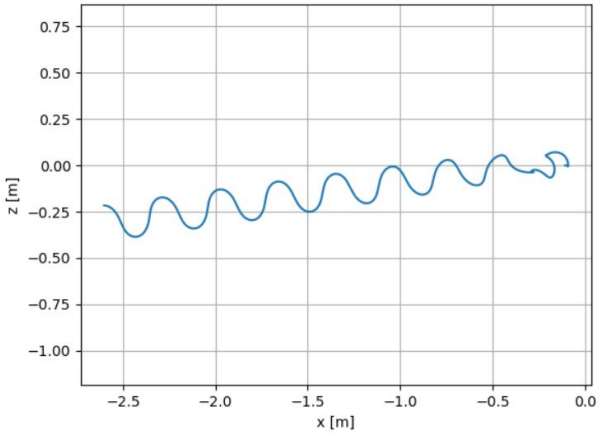


Figure 11: Trajectory when going backward

The robot moves by setting a different phase bias throughout its body so that each joints moves one after the other and not all in the same direction at once. To move backwards, we simply need to input the opposite phase bias to inverse the oscillation movement so that the tail starts the movement instead of the head. This way, instead of pushing with its tail to move forward, the salamander pushes with its head to move backwards. We used a drive of 3.5 for Figure 11.

9e. Cancelled

9f. Limb – Spine coordination

In this next part you will explore the importance of a proper coordination between the spine and the limb movement for walking.

1. Change the drive to a value used for walking and verify that the robot walks
2. Analyze the spine movement: What are your phase lags along the spine during walking? How does the spine movement compare to the one used for swimming?
3. Notice that the phase between limb and spine oscillators affects the robot's walking speed. Run

a parameter search on the phase offset between limbs and spine. Set the nominal radius R to 0.3 [rad]. Include plots showing how the phase offset influences walking speed and comment the results. How do your findings compare to body deformations in the salamander while walking?

4. Explore the influence of the oscillation amplitude along the body with respect to the walking speed of the robot. Run a parameter search on the nominal radius R with a fixed phase offset between limbs and the spine. For the phase offset take the optimal value from the previous sub-exercise. While exploring R, start from 0 (no body bending).

Include plots showing how the oscillation radius influences walking speed and comment on the results.

Answer

2. The phase lag along the spine are set to 0 for a walking behavior. This is because since the legs are making the body move there is no need for a phase lag along the spine to create a movement. This means the whole body moves at the same time in a sinuous manner and uses only the support of the legs to move forward.

3.

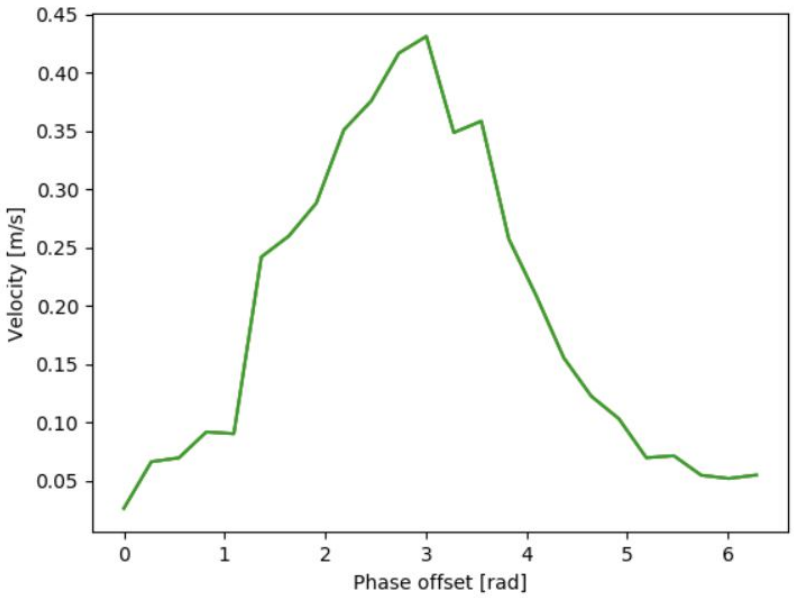


Figure 12: Walking speed depending on phase offset

We ran an experiment testing the robot's walking speed for a phase offset between spine and leg varying between 0 and 2π. We can see the speed has maximum around *pi*. This is because it is the phase offset between legs and spine which correspond to a synchronous movement between the two. The spine and the legs moves together, facilitating movement and increasing the walking speed.

4.

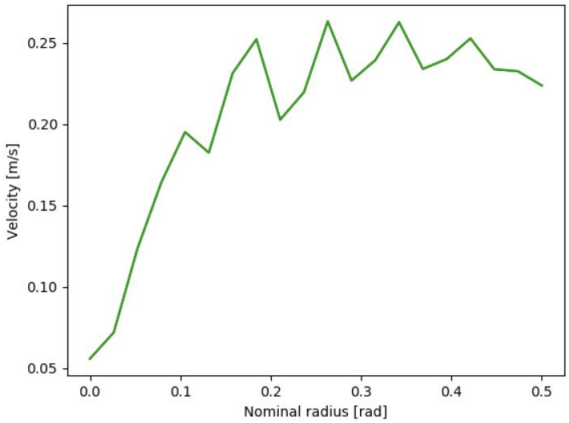


Figure 13: Walking speed depending on Nominal radius

We ran the experiment to see the influence of the oscillation amplitude on the walking speed. We tested with a drive of 1.5 for amplitudes between 0 and 0.5 since anything higher caused saturation or abnormal behavior. We use π for the phase offset value since it is the optimal value.

We can see from Figure 13 that the salamander requires a minimal nominal radius of around 0.1 to move properly. When there is too little bending, the salamander's legs are asynchronous with its body so much that it spends most of the simulation on its belly with its legs moving around above its body.

We can see that some values for nominal radius allow for a faster walking speed. This is once again due to synchronization between the legs and the body, this time reached through the amplitude of the body. The body moves in a way so that each foot is set down when the leg is facing forward, then the body rotates around the leg to move the opposite leg forward. The back legs move in opposite to the front leg, meaning that when the front left foot is down, the back right foot is also down. This gait uses the sinuous movement of the spine to optimize the speed of the walking salamander. Some amplitudes work better than others with the maximal velocity being reached when the front and back legs are in perfect opposite synchronisation, with for example the front right foot and back left foot being set down at the exact same time. Lower velocities are due to the salamander front and back legs being slightly asynchronous.

9g. Land-to-water transitions

1. In this exercise you will explore the gait switching mechanism. The gait switching is generated by a high level drive signal which interacts with the saturation functions that you should have implemented in 9a. Implement a new experiment which uses the x-coordinate of the robot in the world retrieved from a GPS reading (See `self.gps.getValues()` in cmc_robot::log_iteration() for an example). Based on the GPS reading, you should determine if the robot should walk (it's on land) or swim (it reached water). Depending on the current position of the robot, you should modify the drive such that it switches gait appropriately.
2. Run the Webots simulation and report spine and limb angles, together with the x coordinate from the GPS signal. Record a video showing the transition from land to water and submit the video together with this report.
3. (BONUS) Achieve water-to-land transition. Report spine and limb angles, the x-coordinate of the GPS and record a video.

Hint: Use Webots' internal video recording tool to easily record videos.

Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

Answer

2.

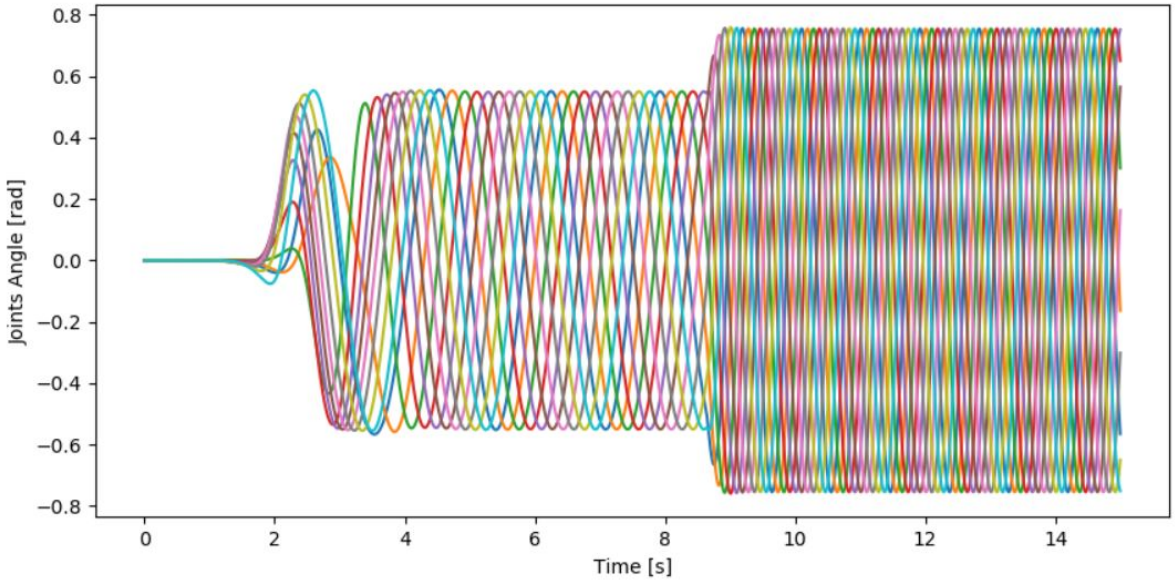


Figure 14: Joints Angles when going from water to land

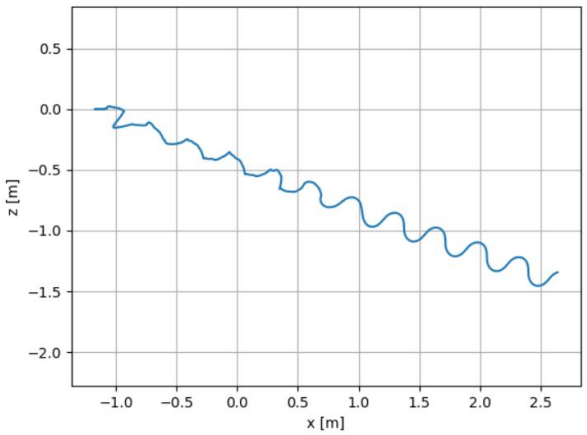


Figure 15: Trajectory when going from water to land

The salamander starts with a drive of 1.5 for walking behavior and when it reaches the water it's drive is changed to 4.0 for a swimming behavior.

The video of the salamander transitioning to the water can be found here : Transition Land to Water

Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

3.

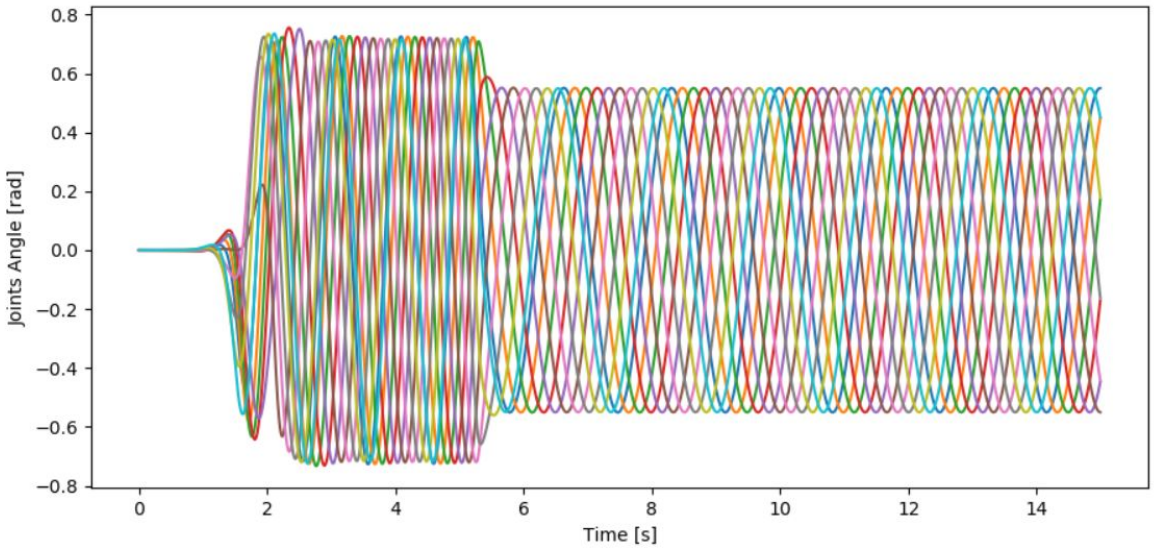


Figure 16: Joints Angles when going from land to water

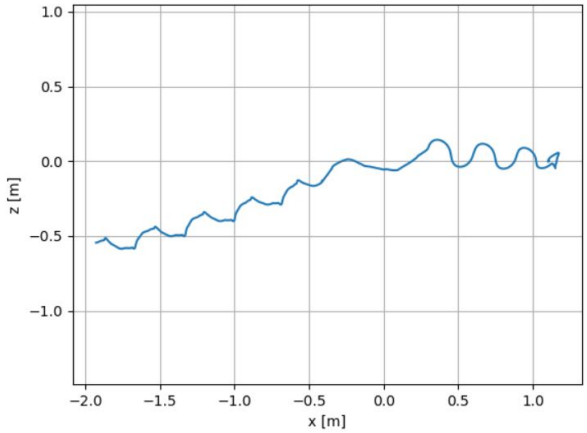


Figure 17: Trajectory when going from land to water

The salamander starts with a drive of 4.0 for a swimming behavior and when it reaches the water it's drive is changed to 1.5 for walking behavior.

The video of the salamander transitioning to land can be found here : Transition Water to Land

Bonnesoeur Maxime
Gautier Maxime
Furrer Stanislas

References

[1] A. Crespi, K. Karakasiliotis, A. Guignard, and A. J. Ijspeert, "Salamandra robotica ii: An amphibious robot to study salamander-like swimming and walking gaits," *IEEE Transactions on Robotics*, vol. 29, pp. 308–320, April 2013.

[2] K. Karakasiliotis, N. Schilling, J.-M. Cabelguen, and A. J. Ijspeert, "Where are we in understanding salamander locomotion: biological and robotic perspectives on kinematics," *Biological Cybernetics*, vol. 107, pp. 529–544, Oct 2013.

[3] A. J. Ijspeert, A. Crespi, D. Ryczko, and J.-M. Cabelguen, "From swimming to walking with a salamander robot driven by a spinal cord model," *science*, vol. 315, no. 5817, pp. 1416–1420, 2007.